Inteligencia Computacional

Práctica 2
QAP

Autor
Abdullah Taher Saadoon AL-Musawi

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación
—
Granada, enero de 2020

Content Index

1- Description

The objective of this Practice is solving Quadratic Assignment Problem (QAP) using Genetic Algorithm, the objective of QAP is to assign n facilities to n locations in such a way as to minimize the assignment cost. The assignment cost is the sum, over all pairs, of the flow between a pair of facilities multiplied by the distance between their assigned locations.

2- Hardware and Software

The following hardware has been used for evaluation and execution of Genetic algorithm to solve Quadratic Assignment Problem:

- ➔ Processor: Intel(R) Core(TM) i7-4702MQ CPU@ 2.20GHz 2.20GHz
- ➔ Installed memory (RAM): 8.00 GB

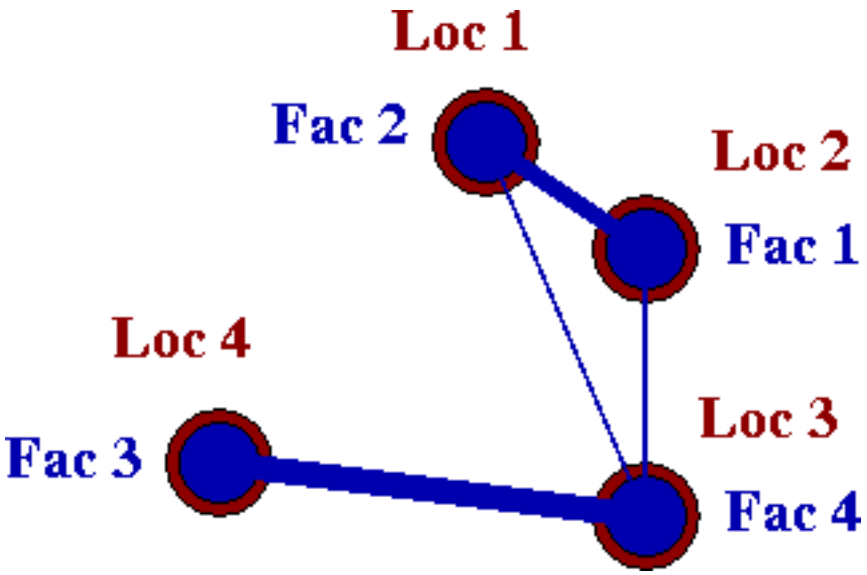
The following was used to develop the necessary software:

- ➔ Programming language: Python 3.6
- ➔ Operating system: Ubuntu 18.04 x64
- ➔ IDE: Visual studio Code
- ➔ Frameworks and libraries
    - Numpy: https://www.numpy.org/

3- Problem Statement

The quadratic assignment problem (QAP) was introduced by Koopmans and Beckman in 1957 in the context of locating "indivisible economic activities". The objective of the problem is to assign a set of facilities to a set of locations in such a way as to minimize the total assignment cost. The assignment cost for a pair of facilities is a function of the flow between the facilities and the distance between the locations of the facilities.

Consider a facility location problem with four facilities (and four locations). One possible assignment is shown in the figure below: facility 2 is assigned to location 1, facility 1 is assigned to location 2, facility 4 is assigned to location 3, and facility 4 is assigned to location 3. This assignment can be written as the permutation p={2,1,4,3}, which means that facility 2 is assigned to location 1, facility 1 is assigned to location 2, facility 4 is assigned to location 3, and facility 4 is assigned to location 3. In the figure, the line between a pair of facilities indicates that there is required flow between the facilities, and the thickness of the line increases with the value of the flow.



To calculate the assignment cost of the permutation, the required flows between facilities and the distances between locations are needed.

Flows between facilities

facility i	facility j	flow(i, j)
1	2	3
1	4	2
2	4	1
3	4	4

Distances between locations

location i	location j	distance(i, j)
1	3	53
2	1	22
2	3	40
3	4	55

Then, the assignment cost of the permutation can be computed as
f(1,2)·d(2,1)+f(1,4)·d(2,3)+f(2,4)·d(1,3)+f(3,4)·d(3,4) = 3·22+2·40+1·53+4·55 = 419.
Note that this permutation is not the optimal solution.

4- Introduction to Genetic Algorithm

Nature has always been a great source of inspiration to all mankind. Genetic Algorithms (GAs) are search based algorithms based on the concepts of natural selection and genetics. GAs are a subset of a much larger branch of computation known as Evolutionary Computation.

GAs were developed by John Holland and his students and colleagues at the University of Michigan, most notably David E. Goldberg and has since been tried on various optimization problems with a high degree of success.

.

CNNs are also being used in image analysis and recognition in agriculture where weather features are extracted from satellites like LSAT to predict the growth and

In GAs, we have a pool or a population of possible solutions to the given problem. These solutions then undergo recombination and mutation (like in natural genetics), producing new children, and the process is repeated over various generations. Each individual (or candidate solution) is assigned a fitness value (based on its objective function value) and the fitter individuals are given a higher chance to mate and yield more "fitter" individuals. This is in line with the Darwinian Theory of "Survival of the Fittest".

In this way we keep "evolving" better individuals or solutions over generations, till we reach a stopping criterion.

Genetic Algorithms are sufficiently randomized in nature, but they perform much better than random local search (in which we just try various random solutions, keeping track of the best so far), as they exploit historical information as well.

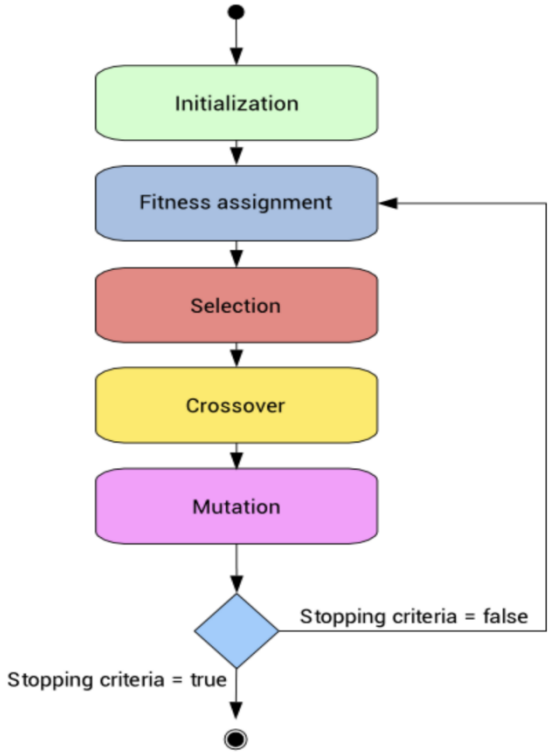
Advantages of GAs

GAs have various advantages which have made them immensely popular. These include −

- Does not require any derivative information (which may not be available for many real-world problems).
- Is faster and more efficient as compared to the traditional methods.
- Has very good parallel capabilities.
- Optimizes both continuous and discrete functions and also multi-objective problems.
- Provides a list of "good" solutions and not just a single solution.
- Always gets an answer to the problem, which gets better over the time.
- Useful when the search space is very large and there are a large number of parameters involved.

5- Implementation

Now, GAs will allow us to obtain a series of generations of individuals that are evolving and improving the previously obtained solutions. The population model is of the generational type, that is, each individual will live for one generation, since in the next generation they will be replaced. a Lamarckian, Balwidian variant will be implemented
The basic process that will generate this algorithm in each iteration is as follows:



5.1 The first thing we have to do it is Initial population.

the generation of individuals is generated randomly in the first iteration. After obtaining these individuals, they are optimized using a local search algorithm and applying the following changes according to the variant of the algorithm used:

Standard : calculate individual fitness

```
for individual in generation:
    individual.fitness = super().calculate_individual_fitness(individual)
```

Lamarckian variant: new individuals resulting from local optimization replace the individual in that population.

```
counter=0
    for individual in generation:
        counter += 1
        individual = super().greedy_optimization( individual )
```

Balwidian variant: Optimizes the individual but only changes its fitness without letting the optimization, thus not allowing the optimization to be in it offspring.

```
counter=0
    for individual in generation:
        counter += 1
        optimized_individual = super().greedy_optimization( individual )
        individual.fitness = optimized_individual.fitness
```

5.2 Selection of individuals

When making the selection of the individuals of the initial population, to later become “parents” two factors are taken into account:

The first factor is a percentage of selection. This is a variable in the algorithm that indicates the percentage of individuals to whom the selection operator will apply. The remaining number of individuals is inserted directly from the initial population. The best individual, the worst, and the best remaining individuals in the population (sorted ascendingly) will be added until the population is completed.

The second factor is Randomly select two different individuals from the current generation and returns the optimal one

```
def binary_tournament(self):
  rand_numb1, rand_numb2 = random.sample(range(0, self.GENERATION_SIZE), 2)

      individ1 = self.current_generation[rand_numb1]
      individ2 = self.current_generation[rand_numb2]

      return min([individ1, individ2])
```

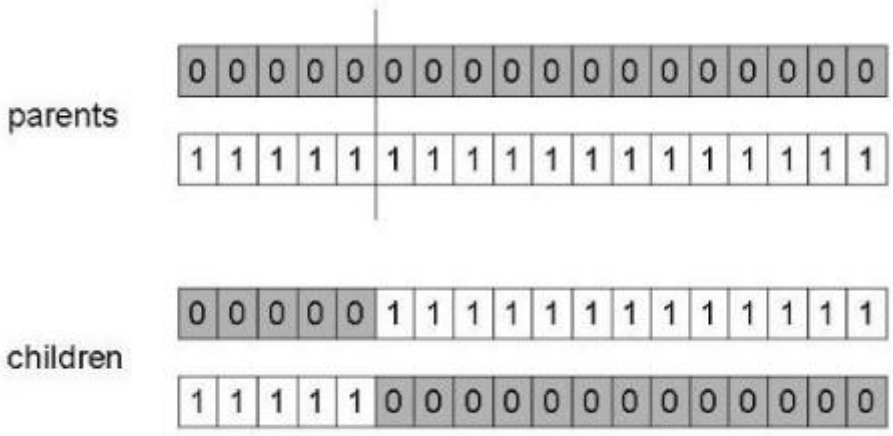
5.3 Generation of parents

```
6    for i in range( self.NUMBER_OF_GENERATIONS ):
7    new_generation = []
8              for j in range( 0, int(self.GENERATION_SIZE), 2 ): # step = 2
9                  parent1 = self.binary_tournament()
```

```
10
11              parent2 = None
12              while parent1 != parent2:
13                  parent2 = self.binary_tournament()
14
```

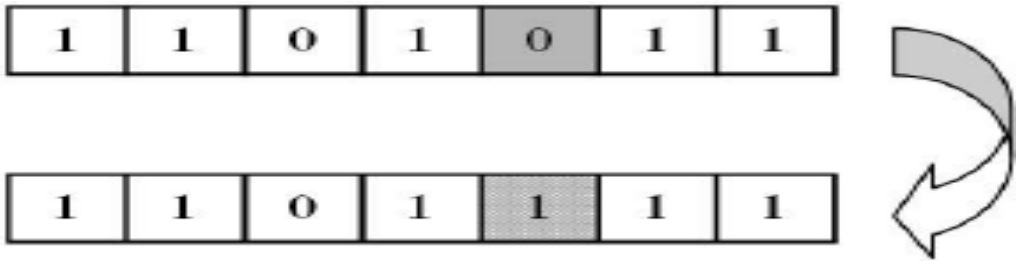
5.4 Crossing individuals

Once the generation of parents has been generated, we proceed to make a cross between them to generate the new generation “daughter”. The crossing operator that has been used is a crossing type at a point, at which a crossing point is selected randomly, the parents are divided at that point and the children are created by exchanging parts of the chromosomes. The following figure shows an example of this type of crossing:



5.4 mutation

This is nothing more than exchanging two values of two positions randomly, this small change implies that a new space is explored of solutions can be better or worse but it will be necessary to generate diversity. For Example



5.5 How to Algorithm works

- ReadFile ()
- Actual_Generation = Create_Genertaion
- Actual_Generation. Calculate_fitness ()
- for i =0.. No_Generation :
- for j =0.. TAM_ Generation:
- Parent1 = binary_tournament(Actual_Generation)
- Parent2= binary_tournament (Actual_Generation)
- son1 , son2 = exchange (Parent1, Parent2)
- son1. mutate ()
- son1. mutate ()
- new_generation .add (son1 , son2)
- best_generation = Actual_Generation.get_best()
- new_generation. replace _ worse_by (best_generation)
- Actual_Generation = best_generation
- Actual_Generation. calculate_fitness ()
- return Actual_Generation. Get_best ()

6- Summary of result

After implementing and executing the three variants of the genetic algorithm. For the results presented in this section the following:

6.1 Standard result

Problem size: 256
Number of generations: 100
Generation size: 60
Fitness of the final best individual: 48274146
Chromosomes:
[199, 60, 65, 163, 138, 223, 147, 227, 160, 182, 237, 47, 149, 248, 209, 120, 16, 201, 195, 70, 67, 132, 177, 73, 238, 192, 137, 37, 20, 240, 145, 220, 78, 22, 232, 109, 119, 15, 112, 2, 167, 69, 203, 143, 107, 246, 185, 42, 141, 51, 134, 155, 190, 234, 191, 170, 126, 34, 74, 151, 41, 12, 241, 180, 186, 96, 117, 7, 228, 99, 61, 64, 98, 35, 40, 101, 9, 130, 93, 104, 87, 216, 26, 14, 245, 28, 210, 204, 174, 235, 38, 17, 249, 85, 21, 166, 25, 162, 194, 75, 188,
139, 159, 211, 152, 153, 187, 198, 148, 164, 233, 39, 124, 53, 213, 86, 81, 1, 205, 57, 146, 247, 4, 95, 102, 253, 165, 59, 231, 77, 62, 31, 215, 200, 219, 193, 239, 202, 18, 183, 244, 58, 49, 46, 243, 169, 226, 108, 118, 54, 83, 89, 222, 197, 56, 68, 0, 255, 218, 63, 150, 196, 94, 224, 48, 172, 175, 127, 8, 92, 176, 136, 114, 154, 254, 173, 251, 156, 236, 144, 111, 90, 208, 157, 43, 250, 229, 179, 10, 135, 113, 82, 52, 110, 225, 33, 214, 45, 189, 6, 100, 105, 55, 66, 29, 30, 128, 11, 221, 27, 72, 212, 103, 32, 230, 13, 23, 91, 44, 184, 121, 50, 3, 24, 252, 116, 140, 36, 242, 79, 76, 106, 168, 84, 115, 5, 129, 171, 131, 206, 158, 161, 88, 80, 217, 142, 97, 71, 122, 133, 19, 123, 207, 178, 125, 181]
Standard executing time: 153.439s

6.2 Baldwinian Result

Problem size: 256
Number of generations: 2
Generation size: 4
Fitness of the final best individual: 52254998
Chromosomes:
[237, 15, 40, 157, 102, 123, 191, 248, 254, 181, 158, 19, 37, 167, 245, 105, 124, 110, 47, 226, 201, 249, 172, 23,
27, 149, 18, 4, 146, 14, 147, 0, 65, 153, 190, 142, 163, 215, 196, 77, 72, 20, 61, 208, 230, 197, 81, 121, 170, 228, 187, 3, 86, 165, 169, 95, 194, 60, 80, 118, 100, 38, 174,

10, 171, 109, 211, 74, 255, 141, 17, 73, 128, 76, 55, 63, 219, 180, 36, 161, 202, 113, 96, 26, 108, 8, 207, 34, 145, 135, 232, 91, 250, 2, 136, 49, 155, 98, 114, 160, 148, 154, 156, 233, 246, 224, 189, 25, 11, 192, 28, 79, 64, 52, 71, 137, 116, 35, 176, 203, 199, 87, 44, 9, 92, 24, 168, 178, 247, 58, 235, 234, 67, 57, 32, 50, 242, 223, 229, 241, 16, 68, 101, 85, 83, 243, 82, 39, 144, 140, 84, 139, 204, 227, 134, 66, 184, 129, 213, 151, 119, 251, 231, 183, 131, 29, 75, 43, 125, 120, 175, 133, 205, 93, 126, 206, 22, 239, 127, 162, 238, 53, 150, 13, 152, 193, 97, 88, 159, 111, 94, 253, 216, 225, 54, 182, 1, 209, 164, 173, 210, 200,
7, 132, 222, 115, 6, 185, 99, 69, 240, 188, 12, 138, 244, 166, 252, 33, 62, 30, 218, 103, 78, 5, 51, 130, 198, 186,
217, 104, 214, 107, 90, 41, 89, 59, 46, 70, 122, 112, 179, 106, 45, 31, 21, 56, 177, 221, 212, 48, 220, 117, 195, 42, 143, 236]
Baldwinian executing time: 383.021s

6.3 Lamarckian Result

Problem size: 256

Number of generations: 2

Generation size: 4

Fitness of the final best individual: 42329844

Chromosomes:

[246, 155, 85, 236, 58, 211, 61, 251, 0, 150, 237, 59, 81, 230, 112, 221, 75, 208, 154, 198, 56, 160, 190, 51, 115, 243, 152, 103, 18, 135, 92, 16, 183, 240, 15, 205, 163, 90, 235, 66, 27, 21, 4, 38, 202, 151, 67, 106, 52, 179, 116, 10, 83, 49, 133, 149, 80, 76, 212, 218, 249, 48, 13, 225, 141, 127, 199, 174, 35, 102, 250, 78, 69, 132, 53, 39, 121, 238, 50, 172, 57, 228, 189, 182, 117, 248, 79, 109, 234, 232, 168, 12, 119, 124, 131, 55, 192, 6, 178, 28, 206,

93, 213, 29, 118, 22, 188, 226, 245, 161, 156, 222, 185, 123, 175, 65, 224, 32, 162, 139, 143, 165, 223, 2, 215, 114, 3, 200, 177, 203, 70, 88, 87, 244, 201, 153, 33, 23,

68, 217, 97, 111, 44, 120, 142, 77, 19, 37, 73, 204, 158, 184, 91, 99, 46, 159, 24, 176, 5, 136, 170, 26, 40, 60, 216, 147, 9, 164, 239, 11, 219, 253, 210, 42, 166, 169, 220, 63, 191, 34, 1, 110, 167, 14, 43, 254, 186, 207, 180, 209, 129, 148, 31, 233, 242, 195, 214, 157, 101, 98, 194, 231, 137, 89, 173, 125, 62, 100, 140, 252, 126, 247, 107, 17, 8, 64, 36, 105, 144, 47, 54, 41, 171, 187, 20, 241, 86, 128, 146, 71, 196, 138, 74, 7, 72, 134, 229, 197, 25, 84, 227, 96, 181, 122, 255, 94, 145, 113, 30, 193, 45, 104, 82, 95, 130, 108]

Lamarckian executing time: 392.962s

7- Bibliography

1 https://neos-guide.org/content/quadratic-assignment-problem

2 https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_introduction.htm

3- https://towardsdatascience.com/genetic-algorithm-explained-step-by-step-65358abe2bf

4- https://medium.com/@himanshubeniwal/handwritten-digit-recognition-using-machine-learning-ad30562a9b64

5- https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3